THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

Schema Mappings for Data Graphs

OPEN ACCESS

Schema Mappings for Data Graphs

Nadime Francis
School of Informatics
University of Edinburgh
nfranci2@inf.ed.ac.uk

Leonid Libkin
School of Informatics
University of Edinburgh
libkin@inf.ed.ac.uk

ABSTRACT

Schema mappings are a fundamental concept in data integration and exchange, and they have been thoroughly studied in different data models. For graph data, however, mappings have been studied in a restricted context that, unlike real-life graph databases, completely disregards the *data* they store. Our main goal is to understand query answering under graph schema mappings – in particular, in exchange and integration of graph data – for graph databases that mix graph structure with data. We show that adding data querying alters the picture in a significant way.

As the model, we use data graphs: a theoretical abstraction of property graphs employed by graph database implementations. We start by showing a very strong negative result: using the simplest form of nontrivial navigation in mappings makes answering even simple queries that mix navigation and data undecidable. This result suggests that for the purposes of integration and exchange, schema mappings ought to exclude recursively defined navigation over target data. For such mappings and analogs of regular path queries that take data into account, query answering becomes decidable, although intractable. To restore tractability without imposing further restrictions on queries, we propose a new approach based on the use of null values that resemble usual nulls of relational DBMSs, as opposed to marked nulls one typically uses in integration and exchange tasks. If one moves away from path queries and considers more complex patterns, query answering becomes undecidable again, even for the simplest possible mappings.

1. INTRODUCTION

Schema mappings are a fundamental notion in data interoperability tasks such as data integration and exchange. They prescribe a virtual or physical transformation of one database into another, and guide notions of query answering over integrated or exchanged data. They have been thoroughly investigated over relational data (see recent books [5, 9, 19] reporting on the developments of the last two

decades). Schema mappings are specified by queries over source and target schemas and some relationship between them, typically containment. For relational mappings, we understand how to build database instances based on them, how to query such instances, and where tractability boundaries lie.

While relational data continues to dominate, there are many other models around, and it is thus natural that efforts have been made to understand schema mappings for other data formats. Notably there were both theoretical investigations and practical implementations of schema mappings for XML, see, e.g., [4,23,33]. While they addressed the same questions as in the relational world, the techniques involved were quite different, as one had to reconcile navigational queries and querying data stored in XML documents.

Another popular data model is graph data. It shows up in many modern applications such as social networks and the Semantic Web, that are naturally modeled as graphs, and where querying graph topology is important. Several industrial strength implementations of graph databases are available, such as Neo4j, Virtuoso, AllegroGraph and others. We refer to [2,3] for surveys of graph data models, languages for them, and implementations. Typical queries for graph data are based on paths or patterns in graphs. In database theory, the most commonly studied classes of queries are based on *regular path queries*, or (RPQs) [16].

When it comes to schema mappings on graph databases, we have much less knowledge compared to relational or XML mappings. Mappings based on containment of path queries were studied in [8, 12–14]. These papers looked into both static analysis questions and answering RPQs and their extensions under such mappings, but the setting as described there falls short of modeling graph databases that occur in practice.

Real-world graph databases mix graph structure with data that resides in nodes and edges. These are referred to as *property graphs* [36]; they form the basic data model of Neo4j and are now advocated by LDBC [27] as the standard model to be adopted by graph database implementations. A theoretical abstraction of property graphs is *data graphs* [30], an analog of data trees that have been actively studied as a similar abstraction of XML documents, cf. [10,37]. In property graphs, each node and each edge can carry a record of values, while in a data graph, just as in data trees, each node contains a single data value. From the point of view of theoretical investigation whose goal is to understand the complexity of the main computational tasks associated with a data model, this abstraction suffices, as property graphs

can be modeled by data graphs, by pushing data from edges to nodes and by creating additional nodes to store multiple data values.

Our main goal is to understand the behavior of schema mappings on data graphs where both navigation and data are taken into account, much as was done for relational and XML data in the past, and to go beyond [8, 12–14] where only navigational aspects have been considered. In doing so, we discover that adding data alters the picture dramatically: some basic tasks quickly become undecidable, and recovering decidability, and especially tractability, requires new techniques that do not mimic those for other data models.

As in [8,12–14], our schema mappings are based on RPQs, but RPQs are applied to data graphs. For such mappings and purely navigational queries based on RPQs and some extensions, query answering is decidable, with coNP data complexity [8,12]; to achieve tractability, regular expressions need to be severely restricted [8]. Answering queries refers to finding *certain answers* that are true in all instances satisfying the conditions of a schema mapping for given source data. This is the standard approach in data exchange and virtual data integration [5, 28].

If graphs contain data, the ability to query it makes a strong impact. For very simple mappings based on RPQs, and very simple queries that combine navigation and data querying, the query answering problem becomes *undecidable*. Even if RPQs used in mappings do not allow any recursion (in essence, we deal with the usual relational schema mappings), answering queries that combine navigation and data is intractable, although decidable.

The undecidability result is rather strong and leaves essentially no hope of having any kind of navigation in mappings except what can be expressed by standard relational schema mappings. It applies to schema mappings whose rules are very simple: they are both *local-as-view* (LAV) and *global-as-view* [28] at the same time, except just one rule which involves the simplest unconstrained reachability query. Moreover the query to be answered is given by a regular expression with equality, which is the simplest kind of regular expressions on paths with data [31].

Thus, either in graph data exchange, or in virtual integration of graph databases, answering queries that refer to data stored in graph nodes is impossible without a dramatic restriction on schema mappings. Such a restriction tells us to forget that we are dealing with a graph database (at least on the target side) and treat it as a relational database instead; then many queries can be answered, albeit with a high complexity.

Having shown these negative results, we need to see what can be recovered. It turns out that we can have efficient query answering for a large class of queries (e.g., all RPQs based on the strongest known class of regular expressions that use data [31]), and do it using the standard route: find a universal solution (cf. [5]) and evaluate the query on it. This works for relational mappings under the additional assumption that the domain of data values contains a special *null* value that behaves as in SQL [17]: no comparison involving a null can be true. This differs from the more common assumption of labeled or marked nulls [5, 19] but is actually more in line with storing graphs in a relational DBMS and uses it for querying.

For the last question addressed in the paper, we look at query languages for graphs that combine navigation and data querying, but are not path-based. In particular, we look at the adaptation of XML language XPath [38] to graphs, as was investigated in [15,30]. Here we show a strong negative result: even if mappings treat graph databases as relational, query answering for graph XPath queries under such mappings is undecidable.

To summarize, our results have the following implications.

1. For the model underlying real-life graph database applications, such as property graphs of Neo4j, schema mappings have to treat graphs as if they were relational databases, i.e., no recursively defined navigational properties are allowed in mappings.

2. Even under such mappings, to achieve tractability of answering path queries in exchange/integration scenarios, one has to populate target instances with null values interpreted as SQL nulls, when it comes to querying them.

3. Moving from RPQs that use data to more general patterns is problematic even for the simplest mappings, as query answering becomes undecidable already for simple fragments of Graph XPath.

Organization Section 2 defines data graph and navigational queries; in Section 3 we describe queries that combine navigation and data querying. Section 4 defines graph schema mappings and querying under them. In Section 5 we prove the main undecidability result for querying under graph schema mappings. Section 6 looks at the restriction of schema mappings that does not allow recursion over targets, and proves decidability and intractability of query answering. Section 7 shows how to restore tractability via null values. Section 8 looks at queries with restricted comparisons, and Section 9 shows undecidability of query answering for Graph XPath. Conclusions are in Section 10. Due to space limitations, we are only able to give sketches and ideas of the proofs.

2. PRELIMINARIES

Graph databases and data graphs For the purposes of navigational querying, graph databases are usually abstracted as labeled directed graphs, i.e., their edges are of the form (v, a, v'), where v, v' are two nodes and a is a label. For data graphs, we follow exactly the same approach except that a node now consists of a node id and a data value.

More precisely, let Σ be a finite set of edge labels, $\mathcal{N}$ a countably infinite set of node ids, and $\mathcal{D}$ be a countably infinite set of data values. A pair $v = (n, d)$ where $n \in \mathcal{N}$ and $d \in \mathcal{D}$ is a *node*. A *data graph* is $G = \langle V, E \rangle$ where V is a finite set of nodes (i.e., $V \subset \mathcal{N} \times \mathcal{D}$) such that no two nodes have the same node id, and $E \subseteq V \times \Sigma \times V$ is a set of labeled edges. For a node $v = (n, d)$, we let $\delta(v)$ refer to the data value d.

A data graph G can also be seen as a relational structure $\langle V, (E_a)_{a \in \Sigma} \rangle$ where each E_a is the binary relation $\{(v, v') \in V \mid (v, a, v') \in E\}$. We will freely move back and forth between these two representations of data graphs.

A *path* π in a data graph G from a node v_1 to a node v_{n+1} is a finite sequence $\pi = v_1 a_1 v_2 \ldots v_n a_n v_{n+1}$, where each v_i is a node of G, each a_i is a symbol of Σ, and (v_i, a_i, v_{i+1}) is an edge of G for each $i \leq n$. We use $\lambda(\pi)$ to refer to the word $a_1 \ldots a_n$, called the *label* of π. The *length* of π, denoted by

$|\pi|$, is the length of the word $\lambda(\pi) = a_1 \ldots a_n$, in this case n. To denote the fact that π is a path from v_1 to v_{n+1}, we will write $v_1 \xrightarrow{\pi} v_{n+1}$. Similarly, for a word $w \in \Sigma^*$, we will write $v_1 \xrightarrow{w} v_{n+1}$ to say that there exists a path going from v_1 to v_{n+1} of label w.

A *data path* is a sequence $d_1 a_1 d_2 \ldots d_n a_n d_{n+1}$ where all the d_is are elements of $\mathcal{D}$ and all the a_js are labels from Σ. Given a path $\pi = v_1 a_1 v_2 \ldots v_n a_n v_{n+1}$ in a data graph $G = \langle V, E \rangle$, the corresponding *data path* is defined as $\delta(\pi) = \delta(v_1) a_1 \delta(v_2) \ldots \delta(x_n) a_n \delta(v_{n+1})$. That is, each node in π is replaced by its data value. Note that data paths are very similar to data words [10, 37] which are words over $\Sigma \times \mathcal{D}$; data paths are data words with one extra data value.

Navigational queries A k-ary *query* q (over Σ) is a mapping that associates with each Σ-labeled data graph $G = \langle V, E \rangle$ a set of k-tuples of nodes $q(G) \subseteq V^k$. We refer to k as the arity of q. We mainly deal with binary queries, i.e., $k = 2$.

The basic queries we consider are *regular path queries (RPQs)*. An RPQ over Σ is just a regular expression e over Σ. Over a (data) graph G, it outputs pairs of nodes connected by a path whose label is in e:

$$e(G) \ = \ \{(v, v') \mid \exists \pi : \ v \xrightarrow{\pi} v' \text{ and } \lambda(\pi) \in e\}$$

A *word* RPQ is an RPQ where e is a word $w \in \Sigma^*$. A *reachability* RPQ is simply $e = \Sigma^*$.

A special case of a word RPQ is an *atomic* RPQ a, where $a \in \Sigma$. Such a query returns the binary relation $E_a = \{(v, v') \mid (v, a, v') \in E\}$.

3. DATA RPQs ON DATA GRAPHS

In this section we review queries on data graphs that combine navigation and data querying. In general, there are three main approaches to such queries. *Data RPQs* are extensions of RPQs with regular expressions that mix alphabet letters and data values [30, 31]; these are the queries we mainly concentrate on as they are natural analogs of navigational queries most commonly studied in graph databases. A different approach is that of *walk logic* [24], initially defined for a slightly different model and then extended to data graphs in [7]. The language is elegant and very expressive, but it was shown to have non-elementary complexity, ruling out its practical applications [7]. Finally, one can have pattern-based languages, like Cypher of Neo4j [36] or analogs of XPath for graphs [15, 30, 36]. We look at those in Section 9.

We now define data RPQs that mix navigation and data. They work on data paths, which are essentially data words; thus, data RPQs will be based on regular languages for data words/paths. The most common automata model that provides the notion of regularity for them is *register automata* [25]. Of course queries are not specified by automata but rather by more declarative regular expressions. While register automata have been around for a while, the study of regular expressions for them is more recent [31], and we shall look at two types of them: expressions with memory, that capture register automata, and weaker expressions with equality that have attractive computational properties. We now define those, specifically for data paths. In all definitions below, we assume a countably infinite set $\mathcal{X}$ of variables (registers).

Regular expressions with memory [30]. These expressions use *conditions* on variables from $\mathcal{X}$ defined by

$$c \ := \ x^= \mid x^{\neq} \mid c \wedge c \mid c \vee c$$

with x ranging over $\mathcal{X}$. Their satisfaction is defined with respect to a pair (σ, d), where σ is a partial map from $\mathcal{X}$ to $\mathcal{D}$ with finite support (i.e., $\sigma : \mathcal{X} \to \mathcal{D} \cup \{\bot\}$ where $\bot$ stands for undefined, and $\sigma(x) \neq \bot$ only for finitely many $x \in \mathcal{X}$), and $d \in \mathcal{D}$. The rules are:

- $\sigma, d \models x^=$ iff $\sigma(x) = d$;
- $\sigma, d \models x^{\neq}$ iff $\sigma(x) \neq d$.

and we use the standard rules for $\wedge$ and $\vee$. Note that conditions are closed under negation; in order to express $\neg c$, we can propagate $\neg$ through $\wedge$ and $\vee$ all the way to basic comparisons and then swap $x^=$ with $x^{\neq}$.

The class $\text{REM}(\Sigma, \mathcal{X})$ of *regular expressions with memory* over alphabet Σ and variables $\mathcal{X}$ is defined by the grammar:

$$e \ := \ \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \bar{x}.e$$

where a ranges over Σ, c over conditions, and $\bar{x}$ over tuples of variables from $\mathcal{X}$. They extend the usual regular expressions with two features: $e[c]$ checks if condition c is satisfied after matching e, and $\downarrow \bar{x}.e$ assigns the current data value to variables in $\bar{x}$ before matching e. To give a couple of examples, the expression $\downarrow x.(a[x^{\neq}])^+$ defines data paths labeled a along which all data values are different from the first one, i.e., data paths of the form $dad_1 a \cdots ad_{n-1} ad_n$ where $d_i \neq d$ for every $i \leq n$. It binds the first data value d, and then checks after reading each letter that the letter was a, and the following data value was different from d. As another example, the expression $\Sigma^* \cdot \downarrow x.\Sigma^+[x^=] \cdot \Sigma^*$ defines data paths in which the same data value occurs more than once: it skips up to that data value, binds it to x, moves one or more labels forward, and checks if the same data value occurs. As usual, Σ^* is $\varepsilon + \Sigma^+$.

The semantics is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where $e \in \text{REM}(\Sigma, \mathcal{X})$, w is a data path, and σ, σ' are variable assignments. Intuitively, it says that if one starts with assignment σ and parses w according to e, then one ends up with σ'. The language of e is then defined as:

$$L(e) \ = \ \{w \mid \exists \sigma : (e, w, \bot) \vdash \sigma\}$$

where $\bot$ is the initial empty assignment (i.e., each variable is undefined). That is, $L(e)$ gives all the data paths that can be parsed according to e without any prior variable binding.

Relation $\vdash$ is defined inductively on the structure of expressions. It uses the notion of *concatenation* of two data paths $w = d_1 a_1 \ldots a_{n-1} d_n$ and $w' = d_n a_n \ldots a_{m-1} d_m$ that share the last and the first data value, defined as $w \cdot w' = d_1 a_1 \ldots a_{n-1} d_n a_n \ldots a_{m-1} d_m$.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = d$ for some $d \in \mathcal{D}$ and $\sigma' = \sigma$.
- $(a, w, \sigma) \vdash \sigma'$ iff $w = d_1 a d_2$ and $\sigma' = \sigma$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff $w = w_1 \cdot w_2$ for some data paths and there exists σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^+, w, \sigma) \vdash \sigma'$ iff $w = w_1 \cdots w_m$ for some $w_1, \ldots, w_m$ and there exist $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_m = \sigma'$ such that $(w, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \leq m$.

- $(\downarrow \bar{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\bar{x}=d}) \vdash \sigma'$, where d is the first data value of w and $\sigma_{\bar{x}=d}$ modifies σ by assigning value d to all variables in $\bar{x}$.

- $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Regular expressions with memory have the same expressiveness as register automata and share many computational properties with them: for example, their nonemptiness problem is Pspace-complete, and membership problem is NP-complete [18, 31]. They are closed under union, intersection, concatenation, Kleene star, but not under complement.

Remark The definition of REM needs to exclude pathological cases $e[c]$ where either c uses a variable that has not been bound to a value, or e is equivalent to ε. This is a small technicality that does not affect us at all as we shall never encounter such expressions. We refer to [30] for details.

Regular expressions with equality This particularly simple class of expressions allows us to capture many useful properties expressed by register automata. The class REE(Σ) of *regular expressions with equality* is defined by:

$$e \ := \ \varepsilon \ | \ a \ | \ e+e \ | \ e \cdot e \ | \ e^+ \ | \ e_= \ | \ e_{\neq}$$

where a ranges over alphabet letters. The language $L(e)$ of data paths denoted e is given as follows:

- $L(\varepsilon) = \{d \mid d \in \mathcal{D}\}$.
- $L(a) = \{dad' \mid d, d' \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e_=) = \{d_1 a_1 \dots a_{n-1} d_n \in L(e) \mid d_1 = d_n\}$.
- $L(e_{\neq}) = \{d_1 a_1 \dots a_{n-1} d_n \in L(e) \mid d_1 \neq d_n\}$.

For example, the language of data paths on which the same data value occurs more than once is given by $\Sigma^* \cdot (\Sigma^+)_= \cdot \Sigma^*$.

While these expressions are strictly weaker than register automata, they have attractive computational properties: for example, both nonemptiness and membership are solvable in Ptime. They have the same closure properties as expressions with memory except they are not closed under intersection [30, 31].

We shall also look at particularly simple expressions, called *path with tests*, defined by $e \ := \ a \ | \ e \cdot e \ | \ e_= \ | \ e_{\neq}$. These are just words, where some subwords carry an annotation indicating whether data values at the beginning and the end of the subword are the same. For example, $(a(bc)_=)_{\neq}$ matches data paths $d_1 a d_2 b d_3 c d_2$ with $d_1 \neq d_2$.

Queries. As queries, we consider RPQs given by regular expressions of one of the above classes. The output $e(G)$ is the set of pairs (v, v') of nodes such that there is a path π in G going from v to v' and with $\delta(\pi) \in L(e)$.

If it is not important which class the expression comes from (with memory or with equality), we use a generic term *data RPQs*. When we want to be more specific, for REMs we refer to *memory RPQs*, for REEs to *equality RPQ*, and for paths with tests, to *data path queries*.

4. SCHEMA MAPPINGS AND QUERY ANSWERING

We now define schema mappings for graph databases, query answering based on them, and explain how standard data exchange and virtual data integration scenarios for graph data are captured.

From now on, we adopt the convention that queries used in mappings will be denoted by lower case letters q, q' etc., and queries that one needs to answer will be denoted by upper case letters Q, Q' etc.

Schema mappings for data graphs As in [8, 12, 14, 21], we define mappings based on RPQs. Assume that we are given two sets of edge labels Σ_s and Σ_t, called *source* and *target* alphabets respectively.

DEFINITION 1. *A* graph schema mapping (GSM) $\mathcal{M}$ *is a set of pairs of RPQs* (q, q'), *where q is an RPQ over Σ_s and q' is an RPQ over Σ_t.*

If $G_s = \langle V_s, E_s \rangle$ and $G_t = \langle V_t, E_t \rangle$ are data graphs over Σ_s and Σ_t respectively, we write $(G_s, G_t) \models \mathcal{M}$ if $q(G_s) \subseteq q'(G_t)$ for every $(q, q') \in \mathcal{M}$, and call G_t a solution *for G_s under $\mathcal{M}$.*

This looks just like the standard definition of schema mappings for graph databases [8, 14]; the difference is that now we deal with *data* graphs, whose nodes are pairs $v = (n, d) \in \mathcal{N} \times \mathcal{D}$. Thus, if $\big((n, d), (n', d')\big) \in q(G_s)$, we require that $\big((n, d), (n', d')\big) \in q(G_t)$. Hence it is not just node ids that must appear in the target graph but also data values that they carry.

A GSM $\mathcal{M}$ is called LAV, or *local-as-view*, if for every pair $(q, q') \in \mathcal{M}$, query q is atomic, i.e., just a letter $a \in \Sigma_s$. These mappings are most commonly used in virtual data integration, and we shall see that most negative results are already true under the LAV restriction. When we say that a LAV mapping is defined using a certain query language, it means that it consists of pairs (a, q') where $a \in \Sigma_s$ and q' belongs to this language. Similarly, a GSM $\mathcal{M}$ is called GAV, or *global-as-view*, if for every pair $(q, q') \in \mathcal{M}$, query q' is atomic.

Query answering As is standard [5, 28], we shall adopt the certain answers semantics for query answering under schema mappings.

DEFINITION 2. *Given a data graph G_s over Σ_s, a GSM $\mathcal{M}$ over Σ_s, Σ_t, and a query Q over Σ_t, the set of* certain *answers to Q on G_s under $\mathcal{M}$ is defined as:*

$$\square_{\mathcal{M}}(Q, G_s) = \bigcap \{Q(G_t) \mid (G_s, G_t) \models \mathcal{M}\}$$

In other words, we look at answers true in all Σ_t data graphs G_t that, together with the source G_s, satisfy conditions of mapping $\mathcal{M}$.

This gives us the main computational problem we study, *query answering under graph schema mappings*. We look primarily at data complexity of the problem when the mapping $\mathcal{M}$ and the query Q are fixed.

PROBLEM :	QUERYANSWERING_GSM($\mathcal{M}, Q$)
INPUT :	A data graph G_s over Σ_s a tuple $\bar{v}$ of nodes of G_s.
QUESTION :	Is $\bar{v}$ in $\square_{\mathcal{M}}(Q, G_s)$?

When $\mathcal{M}$ and Q are clear from the context or not important, we simply refer to *data complexity* of QueryAnswering_GSM. When we speak of *combined complexity*, we assume that Q and $\mathcal{M}$ are part of the input.

Data exchange and virtual data integration We now explain how the setting described earlier subsumes graph data exchange and virtual data integration.

In relational data exchange, one has source and target schemas σ and τ, and schema mappings $\mathcal{M}$ are typically specified by *source-to-target tgds (st-tgds)* $\forall \bar{x} \ (\varphi_\sigma(\bar{x}) \rightarrow \exists \bar{z} \ \psi_\tau(\bar{x}, \bar{z}))$, where $\varphi_\sigma(\bar{x})$ is a query over σ and $\psi_\tau(\bar{x}, \bar{z})$ is a query over τ (usually both are conjunctive). In this case if q is given by φ_σ and q' by $\exists \bar{z} \ \psi_\tau(\bar{x}, \bar{z})$, the above st-tgd just says that for source and target instances S and T satisfying the mapping we must have $q(S) \subseteq q'(T)$. For answering queries Q over the target for a given source database D_s, one defines certain answers as $\square_{\mathcal{M}}(Q, D_s) = \bigcap \{Q(D_t) \mid (D_s, D_t) \models \mathcal{M}\}$, cf. [5]. Thus, for graph schema mappings, we mimic the relational case: graph databases are viewed as relational structures, but we use RPQs instead of conjunctive queries.

Under LAV, the setting also subsumes virtual data integration of graph databases [13, 14, 21]. In general, in data integration, we have sources $S_1, \ldots, S_n$, and a global schema γ; the mapping $\mathcal{M}$ is given by queries $q_1, \ldots, q_n$ over γ which is satisfied by an instance D of γ, written as $((S_1, \ldots, S_n), D) \models \mathcal{M}$, if $S_i \subseteq q_i(D)$ for all $i \leq n$. For a query Q posed against a database of schema γ, virtual data integration defines answers as $\square_{\mathcal{M}}(Q, (S_1, \ldots, S_n)) = \bigcap \{Q(D) \mid ((S_1, \ldots, S_n), D) \models \mathcal{M}\}$, see [28].

Query answering over LAV GSMs corresponds precisely to virtual integration of several graph databases. Indeed, we can view these source graphs as relations E_a for a virtual graph database G. Then our setting – that is, the notions of LAV schema mappings and query answering – is precisely the same as the virtual data integration scenario described above.

5. UNDECIDABILITY OF DATA RPQs

The problem of query answering under graph schema mappings has been studied for purely navigational queries. For mappings given by RPQs, finding $\square_{\mathcal{M}}(Q, G)$ can be done in coNP if Q is an RPQ, or belongs to one of several navigational languages that extend RPQs (conjunctive RPQs, nested regular expressions, or conjunctive nested regular expressions); the problem is coNP-hard even when Q is an RPQ [8, 12].

We show here that the situation is completely different for queries that combine data and navigation: query answering under schema mappings becomes *undecidable* for data RPQs. In fact we show that undecidability holds under three strong restrictions:

- queries are equality RPQs, i.e., are based on the simplest class of data RPQs;

- mappings are local-as-view; and

- queries in mappings are either atomic or the simplest reachability queries Σ_t^*.

We start by introducing a new concept that will help us describe the class of mappings for which the query answering

answering problem is undecidable, and will also be will be important for restrictions that let us regain decidability.

DEFINITION 3. *We call a mapping $\mathcal{M}$ relational if for each $(q, q') \in \mathcal{M}$, the query q' is a word RPQ, i.e., is given by a word $w_t \in \Sigma_t^*$.*

The reason for the name is that queries over Σ_t in these mappings are relational conjunctive queries. A LAV relational mapping consists of pairs (a, w_t) with $a \in \Sigma_s$ and $w_t \in \Sigma_t^*$.

LAV relational mappings eliminate all recursion in queries that define them, and for such mappings, based on what we know from relational data exchange [5], we would expect many query answering tasks to be decidable (more on that in the next section). So what is the smallest addition to such mappings that cannot be expressed with first-order relational queries? Such an addition is the simplest form of reachability, namely reachability by an arbitrary path, in which we do not impose any restriction at all on label use.

To capture this, we define *relational/reachability* mappings $\mathcal{M}$ as those where in all pairs $(q, q') \in \mathcal{M}$, either q' is a word RPQ, as in relational mappings, or the simplest reachability query Σ_t^*. In particular, in a LAV relational/reachability mapping, pairs of queries are of two kinds: either (a, w_t), with $a \in \Sigma_s$ and $w_t \in \Sigma_t^*$, or (a, Σ_t^*).

We can make it even simpler and impose the restriction that every rule in a mapping, except for reachability, be both LAV and GAV (global-as-view [28]). In such a *LAV/GAV relational/reachability mappings*, pairs of queries are of two kinds:

- (a, b) with $a \in \Sigma_s$ and $b \in \Sigma_t$, and

- (a, Σ_t^*) with $a \in \Sigma_s$.

Already for these very simple mappings, answering the simplest data RPQs is undecidable.

THEOREM 1. QueryAnswering_GSM *is undecidable in data complexity for data RPQs.*
In fact, there exists a LAV/GAV relational/reachability mapping $\mathcal{M}$, and an equality RPQ Q such that QueryAnswering_GSM$(\mathcal{M}, Q)$ *is undecidable.*

Proof idea. The proof is rather involved; we now explain the key idea of the encoding and illustrate the main gadgets used in the proof. We reduce from the Post Correspondence Problem (PCP), by building a GSM $\mathcal{M}$ and a query Q from the alphabet such that for each PCP instance, there exist a source database G_s and a pair of nodes $(start, end)$ of G_s such that $(start, end) \notin \square_{\mathcal{M}}(Q, G_s)$ if and only if the PCP instance is satisfiable. We assume that PCP instances are sets of n pairs of nonempty words (referred to as tiles) $(u_r, v_r)_{1 \leq r \leq n}$ over $\Sigma = \{a, b\}$. Both alphabets Σ_s and Σ_t are $\{a, b, i, t, m, \bar{m}, id, s, v, \leftrightarrow, \#\}$. Symbols other than a and b are separators: i, s and v mark what we call the input, solution and verification parts of the encoding; t marks the start of the encoding of a tile and $\leftrightarrow$ marks the separation between words u_r and v_r of a given tile (u_r, v_r); m and $\bar{m}$ designate a specific tile in a list of tiles; id will be used to insert additional data values along paths; and $\#$ marks the end of the encoding.

The source database G_s built from the PCP instance is shown below.
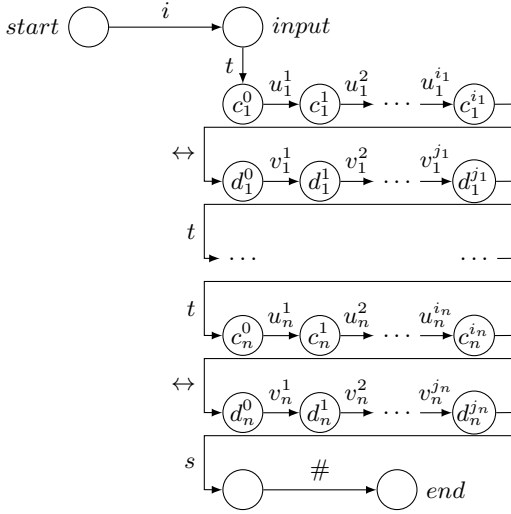
In the figure, the u_i^js and v_i^js are letters of u_i, v_i, and all data values c_i^j and d_i^j are distinct.

The mapping $\mathcal{M}$ has rules (ℓ, ℓ) for $\ell = a, b, t, i, s, \leftrightarrow$ and $(\#, \Sigma_t^*)$. Under this mapping, G_s will be copied into a target graph G_t, but some path can be inserted in place of the $\#$ edge of the input. Now our query will ensure that this path is of a special shape. It will start with an s-edge and then have an encoding of the paths $\pi_{r_1}, \ldots, \pi_{r_m}$ corresponding to the solution of the PCP instance, i.e., a sequence $r_1, \ldots, r_m$ of indexes from $\{1, \ldots, n\}$ (if a solution exists). Then, after a v separator, there is a verification section with labels forming the path $u_{r_1} \ldots u_{r_m}$ (which is the same as $v_{r_1} \ldots v_{r_m}$).

For each r among $r_1, \ldots, r_m$, the path π_r looks as follows.
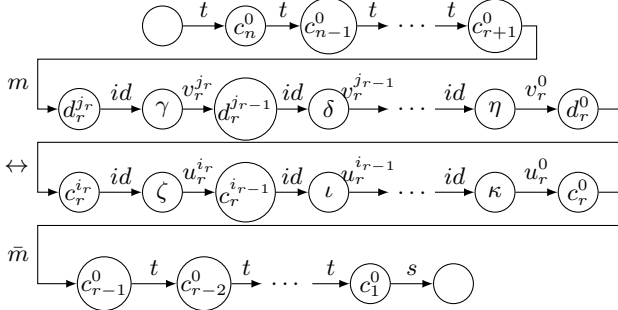
The path π_r starts and ends with the encoding of r split into paths of $n-r$ and r nodes carrying distinct data values, and connected with t-edges. In between, we have encodings of v_r and u_r in reverse order, with extra nodes inserted as additional ids for the purpose of verifying that the solution is correct. These are the nodes connected with id edges, and whose data values are shown as Greek letters in the picture. These four sections of the encoding of π_r are separated with $m, \leftrightarrow$, and $\bar{m}$ letters.

The intuition behind the query Q is that it works on a target instance described above and checks for *errors*, that is, properties that tell us that it is not a correct encoding of a PCP instance and its solution. Then having $(\text{start, end}) \notin \Box_{\mathcal{M}}(Q, G_s)$ implies that there exists a target instance in which none of the errors occur, and thus implies satisfiability of the PCP instance.

There are several kinds of errors, and to ensure that we do have an encoding of PCP, we need to eliminate all of them.

For this, the query Q will be a disjunction of several expressions. One of them, that does not have any references to data values (and thus is an ordinary regular expression) ensures that the path from *start* to *end* is shaped as explained above (rather, it needs to check for errors, i.e., that the path is not shaped as described above, but in this case we have a usual regular expression and can take complement).

Next one can show how to write REE expressions whose negation ensures the following. First, each subpath between two s labels is the encoding of some pair (u_i, v_i) from the input. Thus, the subpath between the first and last s labels is the encoding of a sequence of tiles of the PCP instance.

Next, we use REE expressions to deduce that the subpath that starts after label v only contains pairwise distinct data values, and that the sequence of data values that occur after each label id in the encoding of the solution on the left-hand (resp. right-hand) side of each tile is a copy, in the reverse order, of the sequence of data values in the verification path.

Finally, we use another REE expression whose negation ensures that these data values correspond to the same letter, i.e. it is satisfied if it can find a mismatch. Thus, the sequence of letters on the right-hand side of the tiles in the solution path is the same as the sequence of letters on the left-hand side of the tiles. Thus, the subpath between the first and last s labels is indeed the encoding of a solution to the PCP instance, from which we deduce that it is satisfiable if $(\text{start, end}) \notin \Box_{\mathcal{M}}(Q, G_s)$.

For the other direction, if there is a PCP solution, we build a single-path instance G_t as described above, and then it is a simple matter of checking that $(\text{start, end}) \notin Q(G_s)$. $\square$

Since LAV/GAV relational/reachability mappings represent the simplest possible non-relational addition to the simplest imaginable mappings, this result strongly suggests that for data exchange and integration tasks, target graph databases ought to be viewed as relational databases, and mappings ought to exclude any form of recursive navigation over target data.

6. RELATIONAL MAPPINGS FOR DATA GRAPHS

Given the conclusion of the previous section, we now restrict our attention to relational graph schema mappings in which no recursion over the target schema is allowed. These are GSMs $\mathcal{M}$ (see Definition 3) in which for all $(q, q') \in \mathcal{M}$, the query q' is a word RPQ.

One might be tempted to think that these can be cast as the usual relational mappings over relational representations of graphs, and then results about them can be obtained from what we know in the relational case. While the premise is true, the conclusion is not: we cannot obtain results for such mappings simply by appealing to the relational case. We first explain why this is so, and then present our results about relational GSMs.

Do relational results suffice? Relational GSMs can be easily modeled by the usual relational mappings on graphs viewed as relational databases. A standard representation of a data graph G over Σ_s as a relational database D_G uses binary relations N^s and E_a^s for each $a \in \Sigma_s$ (and likewise N^t and E_a^t for Σ_t). Each node (n, d) is a tuple in N^s, and for each edge $((n, d), a, (n', d'))$, we have the pair (n, n') in E_a^s. Note that since such databases are over disjoint domains

$\mathcal{N}$ and $\mathcal{D}$, we assume that in addition predicates $\mathcal{N}(x)$ and $\mathcal{D}(x)$ are available to distinguish them.

A relational GSM $\mathcal{M}$ between Σ_s and Σ_t data graph can be expressed by means of the relational schema mapping $\mathcal{M}_{\mathrm{rel}}$ defined as shown below.

- For each pair $(q, w) \in \mathcal{M}$ with $w \in \Sigma_t$, we have an st-tgds $\forall x, y\ q(x, y) \rightarrow q_w(x, y)$, where, for a word $w = a_1 \ldots a_n$, the query $q_w(x, y)$ is the conjunctive query $\exists x_1 \ldots x_{n-1}\ E^t_{a_1}(x, x_1) \wedge E^t_{a_2}(x_1, x_2) \cdots \wedge E^t_{a_n}(x_{n-1}, y)$. Note that query q itself need not be conjunctive or even first-order, as in relational st-tgds.

- For each $(q, q') \in \mathcal{M}$, add an st-tgd $\forall x, y\ N^s(x, y) \wedge \exists z\ q(x, z) \rightarrow N^t(x, y)$ and symmetrically with $\exists z\ q(z, x)$. They say that every node that occurs in the result of a source query must be moved to the target.

- Add a key constraint $\forall x, y, y'\ (N^t(x, y) \wedge N^t(x, y') \rightarrow y = y')$ (i.e., each node id has one data value), and target tgds $\forall x, y\ N^t(x, y) \rightarrow \mathcal{N}(x) \wedge \mathcal{D}(y)$ as well as $\forall x, y\ E^t_a(x, y) \rightarrow \exists z, z'\ (N^t(x, z) \wedge N^t(y, z'))$ for each $a \in \Sigma_t$, saying that every node in the target graph must have a data value.

Such a mapping $\mathcal{M}_{\mathrm{rel}}$ precisely describes the action of $\mathcal{M}$ on relational representations of graphs, as the following straightforward result shows.

PROPOSITION 1. *If $\mathcal{M}$ is a relational GSM and G_s is a Σ_s data graph, then solutions for D_{G_s} under $\mathcal{M}_{\mathrm{rel}}$ are exactly the structures of the form D_{G_t} where G_t is a solution for G_s under $\mathcal{M}$.*

Despite this close connection, we cannot directly derive the complexity of query answering from known relational results. To start with, relational mappings can contain pairs (q, w) where q is a complex reachability query, and thus relational results do not apply to it. Even when all such queries q are conjunctive, we still cannot easily apply known results to $\mathcal{M}_{\mathrm{rel}}$. The reason is that answering first-order relational queries under mappings of relational databases, even without target constraints, is already undecidable, cf. [5]. Data RPQs, on the other hand, go beyond first-order queries. Trying to encode data RPQs in a language for which the complexity of query answering in data exchange is known is also problematic, given the scarcity of such languages. For example, [6] looked at a well behaved (in data exchange) fragment of datalog with negation, but the natural coding of equality RPQs in datalog with inequalities over databases D_G does not fall into that fragment. And for memory RPQs, it is not even clear what fragment of a natural relational language they correspond to (e.g., coding of regular expressions into datalog breaks for them).

When it comes to lower bounds, we cannot directly take advantage of existing techniques such as [20, 32] as they use queries that cannot be expressed with RPQs.

Upper and lower bounds for relational mappings Despite the difficulties with the naive approach to reducing to relational results, we can establish both upper and lower bounds on the complexity of query answering. We start with the upper bound.

THEOREM 2. *For relational GSMs, data complexity of* QUERYANSWERING_GSM *for data RPQs is in* CONP, *and its combined complexity is in* PSPACE.

In fact this follows from a more general result. A *homomorphism* between two data graphs $G = \langle V, E \rangle$ and $G' = \langle V', E' \rangle$ is a map $h : \mathcal{N} \rightarrow \mathcal{N}$ such that for each edge (v_1, a, v_2) in E, the edge $(h(v_1), a, h(v_2))$ is in E', where h is extended to nodes by $h\big((n, d)\big) = \big(h(n), d\big)$. A query Q is closed under homomorphisms if, whenever we have a homomorphism $h : G \rightarrow G'$ and $\bar{v} \in Q(G)$, then $h(\bar{v}) \in Q(G')$. It can be readily checked that under this definition, data RPQs are closed under homomorphisms (in fact, a more general result will be shown in Section 7, Proposition 6). The result then follows from the proposition below.

PROPOSITION 2. *Let $\mathcal{C}$ be a class of queries on data graphs that has* PTIME *data complexity and* PSPACE *combined complexity. Assume that every query $q \in \mathcal{C}$ is closed under homomorphisms. Then, for relational GSMs and queries in $\mathcal{C}$, the problem* QUERYANSWERING_GSM *is in* CONP *in data complexity and in* PSPACE *in combined complexity.*

Proof sketch. We first prove an auxiliary result. Assume that $\mathcal{M}$ is a GSM and G_s is a source graph database such that there exists a number $k > 0$ such that, for every rule $(q, q') \in \mathcal{M}$, the language $L(q')$ is contained in Σ_t^k. If G_t is a solution for G_s, we can show that there is another solution $G'_t \subseteq G_t$ whose size is bounded by $(2k + 1) \cdot |\mathcal{M}| \cdot |G_s|^2$.

With this, the proposition follows, since deciding whether $(x, y) \in \square_{\mathcal{M}}(Q, G_s)$ can be done with the following nondeterministic algorithm: first nondeterministically guess a target database G'_t of size at most $(2|\mathcal{M}|^2 + |\mathcal{M}|) \cdot |G_s|^2$, then check whether $(G_s, G'_t) \models \mathcal{M}$ and $(x, y) \notin Q(G'_t)$. Correctness follows from the fact that the mapping is relational, and thus words in $L(q')$ for each $(q, q') \in \mathcal{M}$ have bounded length, which is $\leq |\mathcal{M}|$. It is easy to check that the algorithm has the required complexity under the assumptions. $\square$

In fact, the proof shows a slightly stronger result: the right-hand side queries in mappings need not be single words but could also be regular expressions of the form $w_1 + \cdots + w_m$, where each $w_i \in \Sigma_t^*$ for $i \leq m$.

For query answering under mappings of relational databases, CONP-hardness is already known for unions of conjunctive queries with inequalities [20, 32]. However, as we noticed earlier, reductions used queries that do not correspond to data path queries (non-recursive data RPQs). Nonetheless, it is not hard to come up with a direct reduction to show the following.

PROPOSITION 3. *There exists a data path query Q and a LAV relational mapping $\mathcal{M}$ such that the problem* QUERYANSWERING_GSM$(\mathcal{M}, Q)$ *is* CONP-*complete.*

For the proof, we use a reduction from 3-colorability; there are some technicalities involved, compared to the relational case, due to a very restricted type of query Q.

The query in Proposition 3 uses three inequalities. A similar relational result in [20] uses three inequalities as well, although it also showed that for unions of conjunctive queries with at most one inequality per disjunct, query answering under mappings based on st-tgds is in PTIME. We can adapt the proof and further lower the complexity guarantee for data path queries.

PROPOSITION 4. *For relational GSMs $\mathcal{M}$ and data path queries that have at most one subexpression of the form $e_{\neq}$, data complexity of* QUERYANSWERING_GSM *is in* NLOGSPACE.

As a final remark, we note that for data path queries, query answering is decidable for arbitrary GSMs, not just relational mappings.

PROPOSITION 5. *If Q is a data path query and $\mathcal{M}$ is an arbitrary GSM, then the problem* QUERYANSWERING_GSM$(\mathcal{M}, Q)$ *is in* CONP.

The idea of the proof is that mapping rules that can produce words of length greater than the length of Q are of no use in detecting whether a tuple is a certain answer, and thus one can cut the mapping to a simpler one that essentially resembles the relational case.

7. TRACTABLE QUERY ANSWERING VIA SQL NULLS

Our next question is how to achieve tractability of query answering under graph schema mappings. A standard solution in the case of relations and XML documents is to find a *universal* solution such that running a query over it gives us certain answers, cf. [5]. The CONP-hardness result of the previous section shows that this approach cannot be used directly even with simple equality RPQs. Nonetheless, we show that under some assumptions on data values in target graphs, it can be recovered.

To explain this approach, we recall that when target instances are built in relational data exchange, they are populated either by constants or *marked nulls* [5]. For instance, if we have a relational st-tgd $\forall x, y \; S(x, y) \rightarrow \exists z \; T(x, z) \wedge T(z, y)$, and a source $\{S(a, b), S(c, d)\}$, then the common way is to construct a target instance with tuples $T(a, \mathbf{N}_1), T(\mathbf{N}_1, b), T(c, \mathbf{N}_2), T(\mathbf{N}_2, d)$, where $\mathbf{N}_1, \mathbf{N}_2$ are marked nulls. That is, for each tuple satisfying $S(x, y)$, a new marked null z is invented, and tuples $(x, z), (z, y)$ are put in T.

Such marked nulls are really nothing but constants that did not occur in the source instance: they are new values for which equality is purely syntactic, i.e., $\mathbf{N}_1 = \mathbf{N}_1$ but $\mathbf{N}_1 \neq \mathbf{N}_2$, $\mathbf{N}_1 \neq a$, etc. Thus we can simply view them as elements of the set $\mathcal{D}$ of data values.

In real-life implementations of SQL relational databases, nulls behave differently: there is just one null value $\mathbf{N}$, and no comparison involving $\mathbf{N}$ can evaluate to true. We now show that with nulls behaving like SQL nulls, tractability of query answering can be restored using universal solutions.

Formally, we expand the domain $\mathcal{D}$ of data values with a single null value $\mathbf{N}$, writing $\mathcal{D}_{\mathbf{N}}$ for $\mathcal{D} \cup \{\mathbf{N}\}$. A node $v = (n, \mathbf{N})$ is called a *null node*. Following SQL's rule for nulls, we do not let any comparisons involving nulls evaluate to true. That is, we modify the evaluation of conditions of memory RPQs over $\mathcal{D}_{\mathbf{N}}$ by:

- $\sigma, d \models x^=$ iff $\sigma(x) = d$ and $\sigma(x), d \neq \mathbf{N}$;
- $\sigma, d \models x^{\neq}$ iff $\sigma(x) \neq d$ and $\sigma(x), d \neq \mathbf{N}$,

and for equality RPQs, we allow equalities and inequalities to be true only when both arguments are different from $\mathbf{N}$. Note that we do not use SQL's three-valued logic; at the

end of the section we explain why we can use the simpler approach without any loss of generality.

With this, for a GSM $\mathcal{M}$ and a Σ_s data graph G_s with data values in $\mathcal{D}$, we define certain answers over graphs with null nodes as:

$$\square_{\mathcal{M}}^{\mathbf{N}}(Q, G_s) = \bigcap \left\{ Q(G_t) \; \middle| \; \begin{array}{c} (G_s, G_t) \models \mathcal{M} \\ \text{and } G_t \text{ over } \mathcal{D}_{\mathbf{N}} \end{array} \right\}$$

This corresponds to the standard data exchange setting [5, 20]: source instances do not have nulls, while nulls can appear in the targets.

How does this notion relate to certain answers $\square_{\mathcal{M}}(Q, G_s)$? Since we increase the domain of values, it is immediate that

$$\square_{\mathcal{M}}^{\mathbf{N}}(Q, G_s) \; \subseteq \; \square_{\mathcal{M}}(Q, G_s) \,.$$

Thus, our approach can be seen as providing an underapproximation of query answering in the general case. What makes it attractive is that such an underapproximation can be found efficiently for data RPQs.

THEOREM 3. *If the domain of data values contains the null value $\mathbf{N}$, then data complexity of answering data RPQs under relational graph schema mappings (i.e., checking if $\bar{v} \in \square_{\mathcal{M}}^{\mathbf{N}}(Q, G_s)$) is in* NLOGSPACE. *Its combined complexity is in* PSPACE *for memory RPQs and in* PTIME *for equality RPQs.*

One has matching NLOGSPACE and PSPACE lower bounds as well, as evaluation of data RPQs is NLOGSPACE-hard in data complexity and PSPACE-hard in combined complexity. This theorem will follow from a more general result that provides an algorithm for query answering, even for a larger class of queries.

Recall that a relational GSM $\mathcal{M}$ over Σ_s and Σ_t consists of pairs (q, w) where q is an RPQ over Σ_s and $w \in \Sigma_t^*$. Given such a mapping and a Σ_s data graph G_s, a Σ_t data graph G_t is a *universal solution* for G_s under $\mathcal{M}$ if it is constructed by the following procedure:

1. compute $\text{dom}(\mathcal{M}, G_s)$ which is the set of all nodes appearing in query results $q(G_s)$, for $(q, w) \in \mathcal{M}$, and add all nodes of $\text{dom}(\mathcal{M}, G_s)$ to G_t;

2. then, for each pair $(q, a_1 \ldots, a_k)$ in $\mathcal{M}$, and each pair of nodes $(v, v') \in q(G_s)$, create fresh null nodes $v_1, \ldots, v_k$ and add the path $v a_1 v_1 a_2 \ldots v_{k-1} a_k v$ to G_t.

Note that all universal solutions are actually unique up to a renaming of node ids. Indeed, the only source of nondeterminism in the procedure is the choice of node ids for the newly created null nodes.

Just like the usual universal solutions in relational or XML data exchange, these solutions have homomorphisms into all other solutions. To show this, we first need to extend homomorphisms to data graphs with null nodes. Recall that a homomorphism between data graphs G and G' without null nodes was defined as a map $h : \mathcal{N} \to \mathcal{N}$ such that for each edge $\big((n_1, d_1), a, (n_2, d_2)\big)$ of G, the edge $\big((h(n_1), d_1), a, (h(n_2), d_2)\big)$ is in G'. We now say that a map $h : \mathcal{N} \to \mathcal{N}$ is a *homomorphism between graphs with nulls nodes* if for each edge $\big((n_1, d_1), a, (n_2, d_2)\big)$ of G, there is an edge $\big((h(n_1), d_1'), a, (h(n_2), d_2')\big)$ in G' such that either $d_i = d_i'$ or $d_i = \mathbf{N}$, for $i = 1, 2$. That is, non-null data values

are preserved, but a homomorphism has the freedom to replace $\mathbf{N}$ with another data value. The lemma below justifies the name *universal*.

LEMMA 1. *If $\mathcal{M}$ is a relational mapping, G_s is a Σ_s data graph, G_t is a universal solution for G_s under $\mathcal{M}$ and G_t' is an arbitrary solution over $\mathcal{D}_\mathbf{N}$ for G_s under $\mathcal{M}$, then there is a homomorphism from G_t to G_t' that is the identity on $\mathrm{dom}(\mathcal{M}, G_s)$.*

Our goal is to compute certain answers $\square_\mathcal{M}^\mathbf{N}(Q, G_s)$ by evaluating a query Q over a universal solution G_t. This strategy works for queries that are preserved under homomorphisms. As before, Q is preserved under homomorphisms if, whenever we have two data graphs G and G' with null nodes and a homomorphism h from G to G', then for a tuple $\big((n_1, d_1), \ldots, (n_k, d_k)\big) \in Q(G)$, there is a tuple $\big((h(n_1), d_1'), \ldots, (h(n_k), d_k')\big) \in Q(G')$, where $d_i = d_i'$ whenever $d_i \neq \mathbf{N}$.

The following is a standard observation that is often made in data exchange scenarios to enable efficient computation of certain answers.

THEOREM 4. *Let $\mathcal{M}$ be a relational GSM, G_s a Σ_s data graph over $\mathcal{D}$, and let G_t be a universal solution for it under $\mathcal{M}$. If Q is a query preserved under homomorphisms, then $\square_\mathcal{M}^\mathbf{N}(Q, G_s)$ is the restriction of $Q(G_t)$ to tuples that do not contain any null nodes.*

Proof. If (x, y) is a pair of constant nodes such that $(x, y) \in Q(G_t)$, then the definition of G_t implies that x and y are nodes of G_s. If G_t' is an arbitrary solution for G_s under $\mathcal{M}$, by Lemma 1, there is a homomorphism from G_t to G_t' such that $h(x) = x$ and $h(y) = y$. Since Q is closed under homomorphism, this implies that $(x, y) \in Q(G_t')$. Thus, $(x, y) \in \square_\mathcal{M}^\mathbf{N}(Q, G_s)$.

Conversely, if $(x, y) \in \square_\mathcal{M}^\mathbf{N}(Q, G_s)$, then x and y are nodes of G_s, and thus are constant nodes. Furthermore, since G_t is a solution, we conclude that $(x, y) \in Q(G_t)$, by the definition of certain answers. $\square$

Theorem 4 gives us an algorithm for finding answers to queries preserved under homomorphism. First we construct a universal solution; for relational mappings, this can be done in NLOGSPACE. Then we evaluate Q on it, and then again in NLOGSPACE eliminate null nodes from the result. Thus, it is the data complexity of Q (as long as it is above NLOGSPACE) that determines the complexity of finding certain answers; for instance, if data complexity of Q is NLOGSPACE (or PTIME), then so is the data complexity of QUERYANSWERING_GSM over data domains containing null value $\mathbf{N}$.

Since the data complexity of data RPQs is in NLOGSPACE [30], the last bit that we need to finish the proof of Theorem 3 is the proposition below.

PROPOSITION 6. *Data RPQs are closed under homomorphisms on data graphs with null nodes.*

Proof sketch. It suffices to show that for a REM e, if we have a data path π and its homomorphic image $\pi' = h(\pi)$, then if $\delta(\pi)$ is in $L(e)$, then so is $\delta(\pi')$. This is done by induction on the clauses of the definition of $L(e)$, using the following claim. For any two data paths $w = d_0 a_1 d_1 \ldots d_{n-1} a_n d_n$ and $w' = d_0' a_1 d_1' \ldots d_{n-1}' a_n d_n'$ such that $d_i = d_i'$ whenever

$d_i \neq \mathbf{N}$, for all REMs e', and for all valuations $\sigma_0, \sigma_0', \sigma$ such that $\sigma_0(x) = \sigma_0'(x)$ whenever $\sigma_0(x) \neq \mathbf{N}$ and $(e, w, \sigma_0) \vdash \sigma$, there exists a valuation σ' such that $(e, w', \sigma_0') \vdash \sigma'$ and $\sigma(x) = \sigma'(x)$ whenever $\sigma(x) \neq \mathbf{N}$. $\square$

REMARK 1. We know that $\square_\mathcal{M}^\mathbf{N}(Q, G_s)$ provides an approximation to $\square_\mathcal{M}(Q, G_s)$, i.e., $\square_\mathcal{M}^\mathbf{N}(Q, G_s) \subseteq \square_\mathcal{M}(Q, G_s)$. But how good is such an approximation? The question is akin to those asked about approximations of certain answers over incomplete databases [22, 29] and, just like in those cases, precise theoretical answers are hard to obtain. For example, in the case of Boolean queries, whenever $\square_\mathcal{M}^\mathbf{N}(Q, G_s) \neq \square_\mathcal{M}(Q, G_s)$ – which is bound to happen due to the complexity mismatch – the approximation misses all the correct answers (since *false* is represented as $\emptyset$ and *true* as the singleton empty tuple $\{()\}$). To see how well such schemes actually perform the best one can do is an experimental study, as for instance was done in [22]. For now we have only studied theoretical aspects of such mappings, but the hope is for good quality approximations, given the fact that we have a minimal extension of the domain of values, with just one element.

REMARK 2. SQL nulls use three-valued logic, with truth values *true*, *false*, and *unknown*, and conditions $d = \mathbf{N}$ and $d \neq \mathbf{N}$ evaluate to *unknown*. Here we did not use the three-valued logic and instead made them evaluate to *false*. While this modification could make difference for some queries, for the language of data RPQs there is no change; this is why we chose the simpler way of handling nulls.

Indeed, let $\mathsf{eval}(c, \sigma)$ be the evaluation of condition c as we defined it (i.e., with both $d = \mathbf{N}$ and $d \neq \mathbf{N}$ being *false*). Let $\mathsf{eval}_{\mathrm{SQL}}(c, \sigma)$ be the three-valued SQL evaluation of conditions. Under this evaluation, both $d = \mathbf{N}$ and $d \neq \mathbf{N}$ are *unknown*, and then *unknown* is propagated through $\wedge$ and $\vee$ by *unknown* $\wedge$ *true* = *unknown* $\vee$ *false* = *unknown*; *unknown* $\vee$ *true* = *true*; *unknown* $\wedge$ *false* = *false*. Then it is straightforward to see that $\mathsf{eval}(c, \sigma) = true$ iff $\mathsf{eval}_{\mathrm{SQL}}(c, \sigma) = true$.

8. QUERIES WITHOUT INEQUALITIES

We now look at the behavior of two different languages for data graphs. First, in this section we look at restriction of data RPQs that forbids inequalities, and show that for them, query answering behaves better, at least for relational GSMs. In the next section we show that in contrast, the language that permits non-path patterns behaves rather badly with respect to query answering.

By REM$_=$ and REE$_=$ we mean restrictions of data RPQs based on regular expressions with memory and equality that only use equality comparisons and disallow inequality comparisons. That is, REM$_=$ queries are the same as memory RPQs, but $x^{\neq}$ is not allowed in conditions, and REE$_=$ queries are the same as equality RPQs, but expressions $e_{\neq}$ are not allowed.

Results of Section 6 tell us that data complexity of answering REM$_=$ and REE$_=$ queries under relational GSMs is in CONP. A better NLOGSPACE bound was given for data path queries in REE$_=$ (see Proposition 4), but not for queries outside this narrow class. We now show that tractable bounds can be obtained for *all* queries in REM$_=$ and REE$_=$ under relational mappings.

The idea here follows along the lines of the ideas of Section 7. We define a new kind of canonical solutions, which

bear close resemblance to the universal solutions of Section 7 and play a very similar role. Formally, given a relational GSM $\mathcal{M}$ over Σ_s and Σ_t and a Σ_s data graph G_s, a Σ_t data graph G_t is a *least informative solution* for G_s under $\mathcal{M}$ if it is constructed by the following procedure:

1. compute $\mathrm{dom}(\mathcal{M}, G_s)$ and add all nodes of $\mathrm{dom}(\mathcal{M}, G_s)$ to G_t;

2. then, for each pair $(q, a_1 \ldots, a_k)$ in $\mathcal{M}$, and each pair of nodes $(v, v') \in q(G_s)$, create fresh nodes $v_1, \ldots, v_k$, with respective fresh data values $d_1, \ldots, d_k$ and add the path $v a_1 v_1 a_2 \ldots v_{k-1} a_k v$ to G_t.

The main point of this definition is that the new nodes that are created during the procedure are populated with fresh and distinct data values instead of nulls as in the case for the universal solutions. Then we can prove an analog of Theorem 4:

THEOREM 5. *Let $\mathcal{M}$ be a relational GSM, G_s a Σ_s data graph and let G_t be a least informative solution for G_s under $\mathcal{M}$. Let Q be an $\mathrm{REM}_=$ or an $\mathrm{REE}_=$ query. Then $\square_{\mathcal{M}}(Q, G_s)$ is the restriction of $Q(G_t)$ to tuples with values in $\mathrm{dom}(\mathcal{M}, G_s)$.*

As before, this leads to an algorithm for computing certain answers. We first construct a least informative solution, which can be done in NLOGSPACE for relational mappings. Then we run the query on it, and remove from the output the tuples that use values outside of $\mathrm{dom}(\mathcal{M}, G_s)$. This yields the following corollary:

COROLLARY 1. *For relational GSMs, data complexity of QUERYANSWERING_GSM is in NLOGSPACE for both $\mathrm{REM}_=$ and $\mathrm{REE}_=$ queries, and its combined complexity is in PSPACE for $\mathrm{REM}_=$ queries and in PTIME for $\mathrm{REE}_=$ queries.*

For arbitrary mappings, query answering of RPQs could be coNP-hard even without data values [12], so eliminating inequality would not give us tractability as in Theorem 1. It is still a valid question whether such restriction gives us decidability, in contrast with Theorem 1. This question appears to be quite hard, and we only have a result for $\mathrm{REE}_=$.

PROPOSITION 7. *For arbitrary graph schema mappings, the problem QUERYANSWERING_GSM for $\mathrm{REE}_=$ queries is in coNP for data complexity, and has elementary combined complexity.*

This result already requires an involved proof which establishes a small model property based on Ramsey theorem. This in particular accounts for the high combined complexity bound (which is quadruple exponential, due to Ramsey bounds). Full details are omitted here due to space constraints.

9. GRAPH XPATH

We now look at a different language for expressing queries on data graphs, that is not restricted to path patterns such as data RPQs. In fact languages for graph data such as Cypher of Neo4j [36] or navigational SPARQL [35] do allow non-path patterns.

As a concrete language to work with, we take Graph XPath (GXPath), that adapts the standard XML path language XPath [38] to work on graphs. This language is rather natural for expressing navigational properties that go beyond path patterns, and we have a good understanding of the expressive power of its fragments [30] and their static analysis [26]. GXPath has also been implemented and tested in a distributed environment [34].

There are several variations of GXPath, and we use a minimalistic one, as we prove an undecidability result: a fragment of core GXPath with data value comparisons, denoted by $\mathrm{GXPath}^{\mathrm{core}}_{\sim}$ (following the convention in the literature on data trees, we use $\sim$ to denote data value comparisons).

In defining the language, we assume that for every label $a \in \Sigma$, the alphabet also contains a label a^- which defines the inverse: $E_{a^-} = \{(v, v') \mid (v', v) \in E_a\}$. That is, each edge can be traversed in either direction. As usual for XPath, expressions include node expressions (denoted by φ, ψ) that select nodes, and path expressions (denoted by α, β) that select pairs of nodes. These are given by mutually recursive definitions below:

$$\alpha, \beta \quad := \quad \varepsilon \mid a \mid a^* \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \alpha_= \mid \alpha_{\neq} \mid [\varphi]$$
$$\varphi, \psi \quad := \quad \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle$$

Given a data graph $G = \langle V, E \rangle$, the semantics of a path expression is $[\![\alpha]\!]_G \subseteq V \times V$, and the semantics of a node expression expression is $[\![\varphi]\!]_G \subseteq V$. Precise definitions of those are in Figure 1. Intuitively, ε defines the set of pairs (v, v), while a defines E_a and a^* defines its transitive closure. Expressions $\alpha \cdot \beta$ and $\alpha \cup \beta$ define composition and union, and $\alpha_=$ (or $\alpha_{\neq}$) gives pairs of nodes defined by α that carry the same (or different) data value. Two types of expressions connect path and node expressions: $\langle \alpha \rangle$ is the first projection of the semantics of α (i.e., nodes from which a path satisfying α can start), and $[\varphi]$ gives pairs of nodes (v, v) satisfying φ. Node expressions are closed under Boolean operations.

Note that this fragment excludes many features of full GXPath, such as negation of path expressions $\neg\alpha$, arbitrary transitive closure α^*, and even simpler features such as checking for a constant data value and intersection of paths. But even for this simple fragment of GXPath, query answering is undecidable in data complexity for the simplest possible mappings.

THEOREM 6. *There is a relational GSM $\mathcal{M}$ that is both LAV and GAV and a $\mathrm{GXPath}^{\mathrm{core}}_{\sim}$ node expression φ so that the problem QUERYANSWERING_GSM$(\mathcal{M}, \varphi)$ is undecidable.*

The problem is undecidable for path expressions as well, since $v \in \square_{\mathcal{M}}(\varphi, G)$ iff $(v, v) \in \square_{\mathcal{M}}([\varphi], G)$. The proof of Theorem 6 is based on the following.

LEMMA 2. *There is a fixed labeling alphabet Σ and a $\mathrm{GXPath}^{\mathrm{core}}_{\sim}$ node expression φ such that the following problem is undecidable: given a data graph G over Σ and a node v, is there a data graph $G' \supseteq G$ such that $v \notin [\![\varphi]\!]_{G'}$.*

Theorem 6 is an immediate consequence of Lemma 2: take $\mathcal{M}$ to be $\{(a, a) \mid a \in \Sigma\}$ (i.e., the source and target alphabets are the same). Then $v \notin \square_{\mathcal{M}}(\varphi, G)$ iff $v \notin [\![\varphi]\!]_{G'}$ for some $G' \supseteq G$.

Proof idea (Lemma 2). We prove a stronger statement. A labeled *tree* is said to have the *non-repeating property* if no

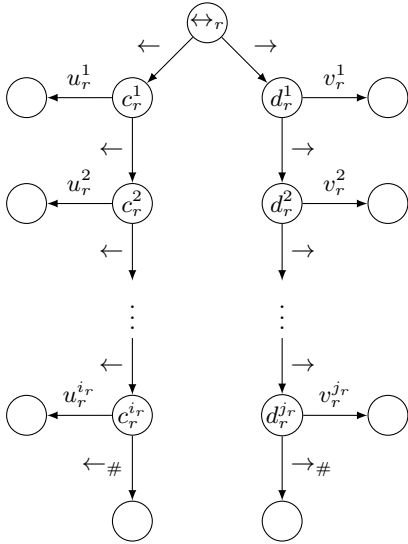| Path expressions		Node expression	
$[\![\varepsilon]\!]_G$	$= \{(v,v) \mid v \in V\}$	$[\![\langle\alpha\rangle]\!]_G$	$= \{v \mid \exists v'\ (v,v') \in [\![\alpha]\!]_G\}$
$[\![a]\!]_G$	$= \{(v,v') \mid (v,a,v') \in E\}$	$[\![\neg\varphi]\!]_G$	$= V - [\![\varphi]\!]_G$
$[\![a^*]\!]_G$	$= \{(v,v') \mid \exists\, v \xrightarrow{\pi} v' \text{ with } \lambda(\pi) \in a^*\}$	$[\![\varphi \wedge \psi]\!]_G$	$= [\![\varphi]\!]_G \cap [\![\psi]\!]_G$
$[\![\alpha \cdot \beta]\!]_G$	$= [\![\alpha]\!]_G \circ [\![\beta]\!]_G$	$[\![\varphi \vee \psi]\!]_G$	$= [\![\varphi]\!]_G \cup [\![\psi]\!]_G$
$[\![\alpha \cup \beta]\!]_G$	$= [\![\alpha]\!]_G \cup [\![\beta]\!]_G$		
$[\![[\varphi]]\!]_G$	$= \{(v,v) \in G \mid v \in [\![\varphi]\!]_G\}$		
$[\![\alpha_=]\!]_G$	$= \{(v,v') \in [\![\alpha]\!]_G \mid \delta(v) = \delta(v')\}$		
$[\![\alpha_{\neq}]\!]_G$	$= \{(v,v') \in [\![\alpha]\!]_G \mid \delta(v) \neq \delta(v')\}$		

Figure 1: Semantics of GXPath$_{\sim}^{\mathrm{core}}$ with respect to $G = \langle V, E\rangle$

alphabet letter can label two different children of the same node. Then Lemma 2 is true under the following assumptions: G is a tree with the non-repeating property, v is its root, all data values in G are distinct, and if there exists $G' \supseteq G$ satisfying $v \notin [\![\varphi]\!]_{G'}$, then such a G' can be chosen to be a tree with the non-repeating property and the same root v.

To show this, as in Theorem 1, we use reduction from PCP. We assume that PCP instances $\{(u_r, v_r) \mid 1 \leq r \leq n\}$ are over the alphabet $\{a, b\}$, and in the encoding we use additional separator labels t, $t_\#$, s, v, $\#$, $\leftarrow$, $\rightarrow$, $\leftarrow_\#$, $\rightarrow_\#$, m, id. Some of them have the same meaning as in the proof of Theorem 1, but new ones are needed to encode the instance in a tree structure: $t_\#$ marks the end of the encoding of a list of pairs from the instance; $\leftarrow$ and $\rightarrow$ mark tree branches that encode u_is and v_is, and $\leftarrow_\#$ and $\rightarrow_\#$ respectively terminate these branches. The PCP instance is encoded as a source tree:
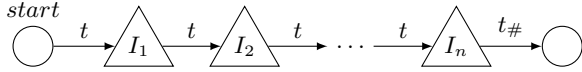
where each I_r encodes the pair (u_r, v_r) as follows:

In the encoding of the entire PCP instance, the "horizontal" path from the *start* node goes via roots of trees I_r. Additionally, data values of all the nodes are pairwise distinct.

The idea is then to construct a node expression φ such that the PCP instance is satisfiable iff there exists a data tree $G' \supseteq G$ rooted at *start* with the non-repeating property satisfying $start \notin [\![\varphi]\!]_{G'}$. The construction uses the idea of attaching the solution and the verification sections to G: the solution section codes a possible solution to the PCP instance and the verification section verifies that it is a solution. The query φ essentially looks for "errors" in the construction of these additional sections. If $start \notin [\![\varphi]\!]_{G'}$, it means there are no errors, and thus a solution was found. The construction is somewhat lengthy and involved, and is given in the appendix. □

Static analysis of GXPath In addition to applications in query answering under GSMs, we can use Lemma 2 to answer some open questions on static analysis of GXPath expressions. Typical static analysis problems are:

- *satisfiability*: given an expression φ, is there a data graph G such that $[\![\varphi]\!]_G \neq \emptyset$?, and

- *containment*: given two expressions φ and ψ, is it true that $[\![\varphi]\!]_G \subseteq [\![\psi]\!]_G$ for every G?

These problems have been studied in depth for XPath on trees, and more recently they were looked at in the context of GXPath, with [26] showing several undecidability results for *regular* GXPath. This more expressive language allows path expressions α^*, i.e., transitive closure of arbitrary path expressions. In our case of *core* GXPath, transitive closure only applies to alphabet labels. For core GXPath, decidability of containment and satisfiability was not known.

We can now use techniques of Lemma 2 to strengthen results of [26] and prove undecidability of static analysis of core GXPath with data value comparisons.

THEOREM 7. *Both satisfiability and containment problems are undecidable for GXPath$_{\sim}^{core}$.*

Proof sketch. We give a proof for the satisfiability problem; undecidability of containment easily follows from it. We use the strong version of Lemma 2 described in the sketch of the proof of Theorem 6. Let Σ and φ be the schema and query fixed by the statement of the lemma. Let G be a data tree with the non-repeating property, all distinct data values, and root v such that if there exists a data graph $G' \supseteq G$ such that $v \notin [\![\varphi]\!]_{G'}$, then such a G' can be chosen to be a tree with the non-repeating property whose root is also v. To prove the theorem, we define a new GXPath$_{\sim}^{core}$ formula φ' that depends on φ and G such that φ' is satisfiable iff there exists a data graph $G' \supseteq G$ such that $v \notin [\![\varphi]\!]_{G'}$.

For this, we define two queries, φ_G and φ_δ, that will ensure that any witness for φ' is a data graph that contains G. The goal for φ_G is to ensure that any data graph satisfying it contains the topological structure of G. The formula φ_G is defined inductively: for a single-node tree it is $\langle \varepsilon \rangle$, and for a tree with children of the root labeled $a_1, \ldots, a_n$ and having subtrees $G_1, \ldots, G_n$, we define φ_G as $\langle a_1 \cdot [\varphi_{G_1}] \rangle \wedge \ldots \wedge \langle a_n \cdot [\varphi_{G_n}] \rangle$.

For φ_δ, let w_x, for a node x of G, be the label of the unique path from the root to x in G. We define $\varphi_\delta = \bigwedge \{ \neg \langle (w_y \cdot (w_y^- \cdot w_z)_=) \rangle \mid y, z \text{ nodes of } G \text{ and } y \neq z \}$. Then one can show that G embeds homomorphically into any graph G' that satisfies φ_G and, if φ_δ in addition holds in G', then G is contained in G', up to a renaming of data values and nodes ids.

With this, we define φ' as $\varphi_G \wedge \varphi_\delta \wedge \neg \varphi$. It is then routine to verify that φ' is satisfiable iff there exists a data graph $G' \supseteq G$ such that $v \notin [\![\varphi]\!]_{G'}$. $\square$

10. CONCLUSIONS

Our main conclusion is that under graph schema mappings, answering queries that combine navigation and data is very different from answering purely navigational queries. Undecidability arises very easily; even under strong restrictions one has intractability of query answering; and achieving tractability requires new techniques that do not mirror those known for relational and XML data.

There are several directions in which we would like to extend this work. First of all, now that we understand what is needed to achieve tractability for the abstraction of data graph, we would like to see how these restrictions look in the case of property graphs of Neo4j [36] and language and constraint formalisms advocated by LDBC [27]. The eventual goal of such a project is to build an integration/exchange system on top of a commercial graph database.

Our results suggest viewing target databases as relational, at least from the point of view of mappings. An alternative of going from relations to graphs has been explored in [11], although mainly intractability results were shown. It would be interesting to see if useful restrictions on relational-to-graph mappings can be found, perhaps using techniques developed here.

We would also like to investigate the use of SQL nulls, as exploited here, in relational data exchange systems. Such systems normally use marked nulls, which need to be implemented on top of relational DBMSs. The advantage of using SQL nulls is that no additional implementation of relational database features is needed, and they appear to allow efficient answering of a rather large class of queries in integration and exchange scenarios.

11. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Angles, C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1): (2008).

[3] R. Angles, M. Arenas, P. Barcelo, A. Hogan, J. Reutter, D. Vrgoč. Foundations of modern graph query languages. arXiv preprint 1610.06264, 2016.

[4] S. Amano, C. David, L. Libkin, F. Murlak. XML schema mappings: data exchange and metadata management. *J. ACM* 61(2): 12:1-12:48 (2014).

[5] M. Arenas, P. Barceló, L. Libkin, F. Murlak. *Foundations of Data Exchange*. Cambridge Univ. Press, 2014.

[6] M. Arenas, P. Barceló, J. Reutter. Query languages for data exchange: beyond unions of conjunctive queries. *Theory of Computing Systems* 49(2):489–564 (2011).

[7] P. Barceló, G. Fontaine, A. W. Lin. Expressive path queries on graphs with data. *LMCS*, 11(4:1) 2015.

[8] P. Barceló, J. Pérez, J. Reutter. Schema mappings and data exchange for graph databases. In *ICDT 2013*, pages 189-200.

[9] Z. Bellahsene, A. Bonifati, E. Rahm, eds. *Schema Matching and Mapping*. Springer, 2011.

[10] M. Bojanczyk. Automata for data words and data trees. In *RTA 2010*, pages 1-4.

[11] I. Boneva, A. Bonifati, R. Ciucanu. Graph data exchange with target constraints. In *EDBT/ICDT Workshop GraphQ*, 2015, pages 171–176.

[12] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi. View-based query processing and constraint satisfaction. *LICS 2000*, pages 361–371.

[13] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.* 64(3): 443-465 (2002).

[14] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Vardi. On simplification of schema mappings. *J. Comput. Syst. Sci.* 79(6): 816-834 (2013).

[15] S. Cassidy. Generalizing XPath for directed graphs. In *Extreme Markup Languages*, 2003.

[16] I. Cruz, A. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.

[17] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

[18] S. Demri, R. Lazic. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).

[19] A. Doan, A. Halevy, Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[20] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[21] N. Francis, L. Segoufin, C. Sirangelo. Datalog rewritings of regular path queries using views. In *ICDT 2014*, pages 107-118.

[22] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *PODS 2016*, pages 211–223.

[23] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, pages 805–810, 2005.

[24] J. Hellings, B. Kuijpers, J. Van den Bussche, X. Zhang. Walk logic as a framework for path query languages on graph databases. In *ICDT 2013*, pages 117–128.

[25] M. Kaminski and N. Francez. Finite memory automata. *TCS*, 134(2):329-363, 1994.

[26] E. Kostylev, J. Reutter, D. Vrgoč. Containment of data graph queries. In *ICDT 2014*, pages 131–142.

[27] LDBC. Linked Data Benchmark Council. http://www.ldbcouncil.org.

[28] M. Lenzerini. Data integration: a theoretical perspective. In *PODS 2002*, pages 233–246.

[29] L. Libkin. SQL's three-valued logic and certain answers. *ACM TODS*, 41(1) (2016), 1–28.

[30] L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *Journal of the ACM* 63(2): 14 (2016).

[31] L. Libkin, T. Tan, D. Vrgoč. Regular expressions for data words. *JCSS* 81(7): 1278-1297 (2015).

[32] A. Mądry. Data exchange: On the complexity of answering queries with inequalities. *IPL*, 94 (2005), 253–257.

[33] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an opensource tool for second-generation schema mapping and data exchange. *PVLDB*, 4 (2011),1438–1441.

[34] M. Nolé, C. Sartiani. A distributed implementation of GXPath. In *EDBT/ICDT Workshops 2016*.

[35] J. Pérez, M. Arenas, C. Gutierrez. nSPARQL: A navigational language for RDF. In *J. Web Sem.* 8(4): 255–270 (2010).

[36] I. Robinson, J. Webber, E. Eifrem. *Graph Databases*. O'Reilly, 2013.

[37] L. Segoufin. Static analysis of XML processing with data values. *SIGMOD Record* 36(1): 31–38 (2007).

[38] XML Path Language (XPath). www.w3.org/TR/xpath.